Enterprise Architektur-Muster

JULIAN BRUDER*, ABDELLAH FILALI*, and LUCA FRANKE*, Hochschule für Technik, Wirtschaft und Kultur Leipzig (HTWK Leipzig), Deutschland

In diesem Papier werden verschiedene Enterprise Architektur-Muster und deren Rolle in modernen Geschäftsprozessen untersucht und anschließend unter Einbeziehung technischer und struktureller Eigenschaften anhand ihrer Agilität bewertet. Dabei orientiert sich die Reihenfolge der Betrachtung jener Architektur-Muster am historischen Verlauf derer Entwicklung und der Notwendigkeit dieser. Genauer werden die monolithische Architektur, modulare monolithische Architektur, service-orientierte Architektur, Microservice-Architektur, Layered-Architecture, Event-Driven Architecture, cloud-native Architektur und die Microkernel-Architektur betrachtet.

Insgesamt zeigt sich, dass klassische Enterprise Architektur-Muster zwar mit geringer initialer Komplexität punkten, mit weiterführender Entwicklung allerdings Flexibilitätsprobleme bedingen. Dem entgegen zeichnen sich die modernen Architektur-Muster durch hohe Agilität und damit hoher Flexibilität gegenüber den in der modernen Geschäftswelt ständig wechselnden Anforderungen aus. Besonders die cloud-native Architektur wird diesen Anforderungen gerecht.

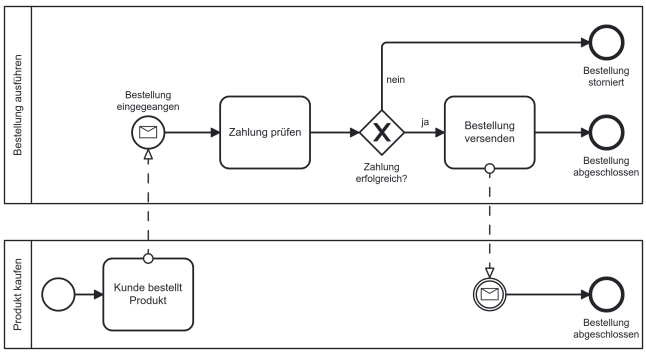

Abb. 1. Geschäftsprozessmodell des MVPs des E-Commerce-Beispiels

1 EINLEITUNG

Moderne Software-Systeme bestehen aus verschiedenen Komponenten [Salah et al. 2016]. Dabei nimmt der Nutzer des Systems jenes zwar als eine Software wahr, jedoch verbirgt sich in Realität meist eine Struktur aus Software-Komponenten und deren Beziehungen hinter dieser Wahrnehmung - ein verteiltes Softwaresystem. Die Art und Weise dieser Struktur und Beziehungen wird als Architektur bezeichnet. Wie wird eine solche Architektur mit all ihren Anforderungen der modernen und schnelllebigen Geschäftswelt konstruiert?

Betrachten wir folgend ein Anwendungsbeispiel zur Motivation. In einem Tech-Startup soll ein Backend für ein internationales E-Commerce-System entwickelt werden. Dabei werden folgende Anforderungen gestellt:

- Zukünftig viele Nutzer und hoher Traffic erwartet
- Geringes Kapital für Infrastruktur
- Rechtliche Regularien sind teilweise unklar
- Hohe Sicherheitsanforderungen aufgrund des Cash-Flows
- Agiles Team von acht fähigen Entwicklern
- Geldgeber wollen erste Auslieferung in zwei Wochen

Das Team arbeitet agil nach Scrum und einigt sich in einer der ersten Planungsphasen auf ein *Minimum Viable Product* (kurz MVP) mit Bestellung, Bezahlung und Versand. Abbildung 1 zeigt die Modellierung des Geschäftsprozesses für diesen MVP. Die Anforderungen sind somit zwar grob strukturiert aber trotzdem ungewiss. Die rechtlichen Regularien sind unklar, was zu späten technischen Änderungen führen kann. Anfangs ist der erwartete Traffic vermutlich niedrig, später sollte dieser aus geschäftlicher Sicht bestenfalls ansteigen. Wie wird der erwartet hohe Traffic aus infrastruktureller Sicht vorgesehen und finanziert? Sind acht Entwickler ausreichend? Kann nach zwei Wochen schon geliefert werden?

All diese Fragen betreffen nicht nur die Software, sondern offensichtlich auch das Geschäft. Die Antworten dafür liefern Enterprise-Architekturen. Deren Ziel ist die Ausrichtung von Geschäft und IT, also die Unterstützung des Geschäfts durch die IT und umgekehrt [Greefhorst and Proper 2011]. Ein Enterprise Architektur-Muster ist dabei eine spezifische Strategie zur Umsetzung dieser Ausrichtung. Erwähnenswert ist hier die Nähe zur Bedeutung des lateinischen Wortes *architectura*. Wörtlich übersetzt ist es die *Wissenschaft der Baukunst* - meint aber sowohl das Produkt des Bauens als auch den Prozess des Bauens [1]. In Bezug dazu kann die Enterprise-Architektur also als Vision einer grundlegenden Struktur und Beziehungen von Komponenten eines Systems betrachtet werden. Wichtig ist hierbei die Abgrenzung zur Software-Architektur, der grundlegenden Struktur und Beziehungen von Teilen einer Software.

Im Verlauf der Arbeit werden verschiedene Enterprise Architektur-Muster und deren Rolle in modernen Geschäftsprozessen untersucht und anschließend unter Einbeziehung technischer und struktureller Eigenschaften anhand ihrer Agilität bewertet. Dabei wird auch immer auf das eingangs erwähnte und in Abbildung 1 dargestellte E-Commerce-Beispiel zurückgegriffen. Die Arbeit ist wie folgt strukturiert:

Zunächst wird in Abschnitt 2 auf die monolithische Architektur eingegangen, bevor die Modularisierung des Monolithen in Abschnitt 3 diskutiert wird. Danach wird in Abschnitt 4 die service-orientierte Architektur betrachtet. Folgend wird in Abschnitt 5 die Microservice-Architektur diskutiert. Anschließend wird in Abschnitt 6 untersucht, wie eine Schichten-Architektur andere, dienstbasierte Architekturen ergänzen kann. Daraufhin werden sowohl die event-basierte Architektur in Abschnitt 7 als auch die cloud-native Architektur in Abschnitt 8 erläutert. Als letztes Architektur-Muster wird in Abschnitt 9 die Microkernel-Architektur betrachtet.

*Alle Studierenden trugen zu gleichen Teilen zu dieser Arbeit bei.

Diese Arbeit wurde im Rahmen des Mastermoduls „Software Engineering“ (Dozent: Prof. Dr. Andreas Both) an der HTWK Leipzig im Wintersemester 2024/2025 erstellt.

[1] https://www.duden.de/rechtschreibung/Architektur, abgerufen am 17.01.2025

Abschließend werden die Architekturen in Abschnitt 10 nach technischen und agilen Kriterien verglichen und folglich die Ergebnisse dieser Arbeit zusammengefasst.

2 MONOLITHIC ARCHITECTURE

Der Begriff *Monolith*, abgeleitet aus dem Altgriechischen und mit der Bedeutung *einheitlicher Stein*[2], wird in der Softwarearchitektur verwendet, um ein Designmuster zu beschreiben, bei dem die gesamte Funktionalität einer Anwendung in einem einzigen, zusammenhängenden System integriert ist. In dieser Architektur übernimmt ein einzelner Prozess die Ausführung der gesamten Anwendung [Chen et al. 2017, 1].

Anwendungen, die diesem Architekturansatz folgen, bestehen aus eng miteinander verknüpften Komponenten, die wechselseitig voneinander abhängen. Diese Komponenten können weder unabhängig betrieben, noch in bestimmten Fällen, isoliert kompiliert werden [Salaheddin and Ahmed 2022, 485].

Diese Architektur bietet insbesondere für kleinere Anwendungen mehrere Vorteile, wie etwa eine vereinfachte Testbarkeit, Logging, Deployment und Debugging. Ein weiterer Vorteil besteht darin, dass keine separate Datenbanksynchronisation erforderlich ist, da sämtliche Daten in einer einzigen Datenbank gespeichert werden [Blinowski et al. 2022, 20358].

Betrachten wir das E-Commerce-Beispiel . Dafür definieren wir drei Klassen (Siehe Abb. 2):

- `OrderService`: Klasse, die die Bestellungen verwaltet
- `PaymentService`: Klasse, die die Zahlungen abwickelt
- `ShipmentService`: Klasse, die die Lieferungen initiiert

Abb. 2. Monolith Architektur

Die Kommunikation zwischen den Klassen erfolgt über Methodenaufrufe. In diesem Zusammenhang stellt die Klasse `OrderService` die zentrale Komponente dar, die die Methoden anderer Klassen nutzt, um den Bestellprozess zu steuern. Ein wesentlicher Vorteil dieses Ansatzes liegt in der einfachen Kommunikation zwischen den Komponenten. Durch den direkten Einsatz von Methodenaufrufen wird die Komplexität verringert, die typischerweise mit der Interaktion zwischen verschiedenen verteilte Systemen verbunden wäre.

Eine beispielhafte Implementierung des `PaymentServices` in einer monolithischen Architektur ist im Anhang A.1 dargestellt. Der vollständige Quellcode des E-Commerce-Beispiels kann zudem bei GitHub[3] eingesehen werden.

Mit dem Wachstum und der zunehmenden Komplexität der Code-Basis treten jedoch einige Nachteile auf. Änderungen an einer Komponente können unerwartete kaskadierende Fehler auslösen, was die Weiterentwicklung in kleinen, autonomen Teams erschwert und verlangsamt.

Ein weiterer Nachteil ergibt sich aus der engen Kopplung zwischen den Komponenten, die durch direkte Methodenaufrufe entsteht. Änderungen an der Implementierung oder der Signatur einer Methode können in der Folge die gesamte Anwendung beeinflussen.

Zudem wird die Wiederverwendung von Funktionalitäten erschwert, da die Komponenten stark miteinander verknüpft sind. Dies führt zu einer Duplizierung von Code, was die Kosten für Änderungen erhöht, da diese an mehreren Stellen vorgenommen werden müssen.

Die daraus resultierende Komplexität und die erschwerte Wartung führen zu längeren Iterationen. Da das Deployment nur als Ganzes erfolgen kann und die Iterationen sich verlängern, kommt es zu seltenen Auslieferungen.

Ein weiteres Problem liegt in der erschwerten horizontalen Skalierung, da die Anwendung nur als Ganzes skaliert werden kann. Dies stellt eine Herausforderung dar, da bei wachsenden Anforderungen die gesamte Anwendung skaliert werden muss, anstatt einzelne Komponenten unabhängig voneinander zu skalieren.

Insgesamt ist die monolithische Architektur eine geeignete Lösung für kleinere Anwendungen, jedoch weniger geeignet für größere Systeme, da sie die Agilität und Flexibilität erheblich einschränken kann.

3 MODULAR MONOLITHIC ARCHITECTURE

Das Hauptproblem der zuvor beschriebenen Architektur liegt in der engen Kopplung der Komponenten, was zu einer erhöhten Komplexität und begrenzten Flexibilität führt. Die Modular Monolithic Architecture stellt eine Weiterentwicklung der Monolithic Architecture dar, indem sie die Vorteile des monolithischen Ansatzes bewahrt und gleichzeitig die Nachteile der engen Kopplung verringert.

In diesem Architekturmodell wird die Code-Basis in mehrere Module unterteilt, die jeweils eine spezifische Teilfunktionalität der Anwendung implementieren.[Su and Li 2024, 11].

Die Modularisierung trägt zudem zur Reduzierung der Komplexität bei und ermöglicht eine bessere Organisation der Code-Basis, was sich positiv auf die Wartbarkeit der Anwendung auswirkt [Barde 2023, 23 - 24].

Betrachten wir erneut das E-Commerce-Beispiel. Diesmal wird die Anwendung in drei Hauptmodule aufgeteilt (Siehe Abb. 3):

- `order`: Modul, das für die Verwaltung der Bestellungen verantwortlich ist
- `payment`: Modul, das für die Abwicklung der Zahlungen verantwortlich ist
- `shipment`: Modul, das für die Initiierung der Lieferungen verantwortlich ist

Abb. 3. Modular Monolith Architektur

[2]https://www.duden.de/rechtschreibung/Monolith, abgerufen am 17.01.2025
[3]https://github.com/Beleg-6-EAP/demo-monolith-ecommerce

Wie Abbildung 3 zeigt, ist jeder Service in einem eigenen Modul gekapselt. Ein konkretes Beispiel hierfür bietet die Implementierung des `OrderServices`, welche im Modul `order` realisiert ist (siehe Anhang A.2). Die Logik des `OrderService` ist vollständig innerhalb des entsprechenden Moduls integriert, während die Kommunikation mit anderen Modulen durch klar definierte Spezifikationen geregelt wird. Für weiterführende Informationen und die vollständige Implementierung des E-Commerce-Beispiels wird auf das GitHub-Repository verwiesen[4].

Durch die Modularisierung sind die Komponenten weniger eng miteinander gekoppelt, was eine verbesserte Zusammenarbeit in kleinen, autonomen Teams fördert, im Vergleich zu traditionellen monolithischen Ansätzen.

Ein Nachteil bleibt jedoch bestehen: Die Anwendung muss weiterhin als Ganzes deployed werden, was die Iterationen verlangsamt. Zudem bleibt die horizontale Skalierung weiterhin erschwert, und die Funktionalitäten können nicht wiederverwendet werden, da sie immer noch Teil einer einzigen Anwendung sind und eng miteinander gekoppelt bleiben.

4 SERVICE-ORIENTED ARCHITECTURE

Bei der (modularen) monolithischen Architektur zeigte sich die enge Kopplung der Funktionalitäten als zentrales Problem. Dieses Problem löst die service-orientierte Architektur (kurz SOA), indem sie den Fokus auf die Nutzung wiederverwendbarer Dienste (englisch *services*) setzt. Ein Dienst ist dabei eine meist grob-granulare Unternehmensfunktionalität.

Betrachten wir im Folgenden die Komponenten einer SOA:

- Service Provider: Komponente, die einen spezifischen Dienst anbietet [Endrei et al. 2004]
- Service Consumer: Komponente, die einen bereitgestellten Dienst nutzt
- Service Registry: Komponente zur Sammlung von Metadaten über Dienste
- Service Bus: Zentrale Komponente, die sowohl als Broker zwischen Providern und Konsumenten fungiert als auch zusätzliche Aufgaben wie beispielsweise Routing übernimmt

Die Beziehungen zwischen diesen Komponenten sind in Abbildung 4 dargestellt. Darin ist zu erkennen, dass Provider und Consumer sich nicht gegenseitig kennen. Damit ein Consumer trotzdem einen Dienst eines Providers nutzen kann, werden Metadaten zum Provider und dessen Dienst in einer Service-Registry gespeichert. Durch diese kann der Service-Bus auf Anfrage eines Consumers für einen spezifischen Dienst einen solchen in der Service-Registry suchen. Wenn ein solcher dort gelistet ist, dann kann der Consumer den Dienst des Providers per Kommunikation über den Service-Bus nutzen. Damit dient dieser offensichtlich als Vermittler zwischen Consumer und Provider, wobei der Consumer die Spezifikation einer Unternehmensfunktionalität nutzt, der Provider diese implementiert und die Registry jene persistiert [Bianco et al. 2007, 16][Endrei et al. 2004, 19-26]. Daher sind Provider und Consumer lose zueinander gekoppelt. Um die Kommunikation am Bus zu vereinheitlichen, wird auf standardisierte Web-Kommunikation zurückgegriffen. Meist werden dafür zum Nachrichtenaustausch das Protokoll SOAP und die Schnittstellenbeschreibungssprache WSDL verwendet.

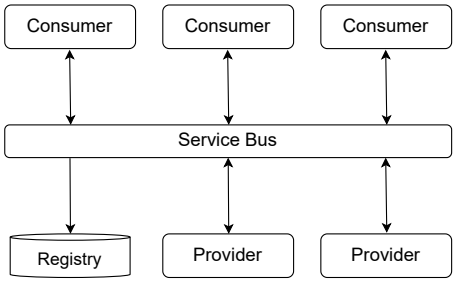

Abb. 4. Aufbau der Service-oriented Architecture

Betrachten wir im Folgenden die Anwendung einer SOA auf das E-Commerce-Beispiel. Dafür definieren wir drei verschiedene Services:

- `OrderService`: Ein Dienst, der den gesamten Bestellvorgang verwaltet
- `PaymentService`: Ein Dienst, der den gesamten Bezahlvorgang verwaltet
- `ShipmentService`: Ein Dienst, der den gesamten Versandvorgang verwaltet

Abbildung 5 stellt die Anwendung der Architektur auf das Beispiel dar.

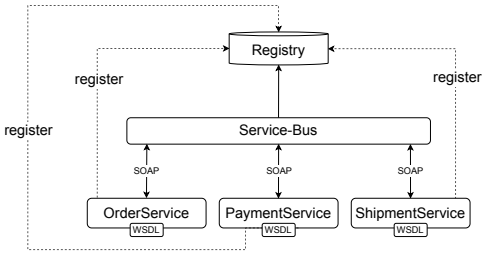

Abb. 5. E-Commerce-Beispiel mit Service-oriented Architecture

Alle drei Dienste agieren dabei offensichtlich als Provider und registrieren ihre Dienste bei der Registry. Der `OrderService` nutzt die Spezifikation der beiden Dienste für Bezahlung und Versand, ist also zusätzlich Konsument. Die Konsumierung der Dienste erfolgt dabei sequenziell, indem der `OrderService` nach Lokalisierung von `PaymentService` und `ShipmentService` erst mit ersterem und dann mit letzterem über den Bus via SOAP kommuniziert.

Da die Provider ihre Dienste dabei mit WSDL spezifizieren und die Registry diese speichert, können die Implementierungen leicht ausgetauscht werden. Im Beispiel könnte dabei möglicherweise ein spezifischer `PaymentService` von PayPal durch einen von Apple Pay ersetzt werden. Wenn sich die Spezifikation jener nicht ändert, dann ändert sich auch die Nutzung der Funktionalität beim Consumer nicht, sodass dieser einfach auf geänderte Anforderungen reagieren kann.

[4] https://github.com/Beleg-6-EAP/demo-modulith-ecommerce

Im Anhang A.7 ist eine beispielhafte Implementierung der Service-Registry zu finden. Die vollständige Implementierung des E-Commerce-Beispiels kann bei GitHub[5] eingesehen werden.

Wie schon bei der modularen Monolithen-Architektur in Abschnitt TODO ermöglicht die lose Kopplung kleine autonome Teams. Im Gegensatz zu jener Architektur können aufgrund der Eigenständigkeit der Services in einer SOA jene allerdings auch individuell pro Team ausgeliefert werden. Das ermöglicht kürzere Iterationen und somit häufigere Auslieferung als beim modularen Monolithen. Dadurch und durch die lose Kopplung kann außerdem flexibler auf wechselnde Anforderungen reagiert werden. Ebenfalls können Entwicklungszeit gesenkt und somit auch Kosten durch die Wiederverwendbarkeit von Diensten eingespart werden werden.

Die meist grobe Granularität der Dienste führt allerdings langfristig zu Abhängigkeiten zwischen Diensten. Aus diesem Grund und der zentralen Abhängigkeit der Dienste an den Service-Bus lassen sich Systeme mit SOA jedoch weiterhin schwer horizontal skalieren.

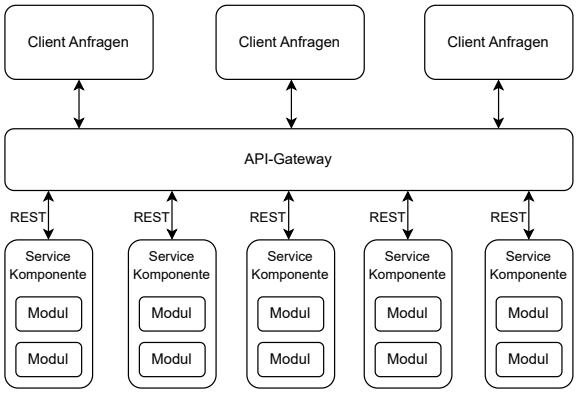

Abb. 6. Aufbau einer Microservices Architecture

5 MICROSERVICES ARCHITECTURE

Die Microservices Architecture greift grundlegend das Konzept einer Service-oriented Architecture auf, legt den Fokus aber auf eine fein granuliertere Aufteilung mit dem Ziel der Isolation.

Die bisher meist grob granulierten Dienste werden auf einzelne Funktionalitäten reduziert und bestenfalls vollständig voneinander isoliert. Im Gegensatz zu SOA greifen die Services nicht mehr auf eine zentrale Persistenz-Schicht zu, sondern haben private, für andere nicht einsehbare Persistenzen[Liu et al. 2020, 2].

Wie in Abbildung 6 dargestellt, werden alle von Clients ausgehenden API-Anfragen zunächst von einem API-Gateway entgegengenommen. Da Anfragen von Clients oft mehrere Services betreffen, wird das Gateway als Vermittler genutzt, um die Anfragen an die entsprechenden Services zu orchestrieren. Dabei profitiert das Gateway von geringen Latenzen aufgrund von physikalischer Nähe zu den Service Komponenten und einer besseren Netzwerkanbindung im Vergleich zum Client. Bei Anfragen an mehrere Services akkumulieren sich diese Ersparnisse und Antwortzeit verkürzt sich insgesamt [Richards 2015, 30]. Da Gateways keine Daten halten, sondern nur Anfragen weiterleiten, sind sie leicht horizontal skalierbar und können bei Bedarf um weitere Instanzen erweitert werden.

Es besteht die Möglichkeit die Service-Komponenten mit unterschiedlichen Technologien zu entwickeln, da die Kommunikation über standardisierte Schnittstellen stattfindet. Diese Schnittstellen sind meist REST-Endpunkte, können aber mithilfe von alternativen Protokollen wie Messaging Queues, RPC oder Event-Streaming implementiert sein. Durch die Aufteilung in isolierte Komponenten, kann die Entwicklung dieser auf kleine, autonome Teams verteilt werden.

Damit ist die Entkopplung der einzelnen Services ein maßgebendes Merkmal dieses Musters. Dies wird dadurch erreicht, dass alle Services nur über ihre definierten Schnittstellen, miteinander kommunizieren können. Die hohe Entkopplung der Komponenten ermöglicht es, Services unabhängig voneinander zu entwickeln, zu testen und auszuliefern. Pipeline- und Deploymentprozesse finden also ebenfalls isoliert voneinander statt, was die Auslieferung der Software beschleunigt [Richards 2015, 27].

Der korrekte Entwurf der Architektur und die darin passend gewählte Granularität der Services ist entscheidend, um die Vorteile der Microservices Architecture voll auszuschöpfen und stellt hier die größte Herausforderung dar. Werden die Komponenten zu groß gewählt, dann könnten die Vorteile der Microservices Architecture nicht mehr greifen, weil innerhalb der Komponenten Abhängigkeiten entstehen, die das isolierte Entwickeln oder Testen von Features verhindern. Auf der anderen Seite führt eine zu geringe Granularität dazu, dass die Anzahl der Services zu groß wird und die Kommunikation einen zu großen Overhead verursacht. Jede Nachricht, die unter den Services ausgetauscht wird, verursacht Latenz, die sich akkumuliert und die Performance des Systems beeinträchtigt. Weiterhin werden Integrationstests durch die hohe Anzahl an Services aufwändiger und der Orchestrierungsaufwand des Gateways steigt an [Richards 2015, 32].

Sollten serviceübergreifende Persistenz-Zugriffe notwendig sein, so könnten diese über eine geteilte Datenbank stattfinden. Dadurch werden ungewollte Abhängigkeiten zwischen Services vermieden und die Kopplung der Komponenten verringert. Dabei ist jedoch darauf zu achten, dass die Datenbank nicht zu einer zentralen Abhängigkeit wird, die die Vorteile der Microservices Architecture wieder rückgängig macht [Richards 2015, 33].

Betrachten wir nun, wie dieses Architekturmuster auf das Beispiel der E-Commerce Anwendung übertragen werden kann. Ein möglicher Aufbau der Anwendung ist in Abbildung 7 dargestellt. Sie ist in drei Services aufgeteilt, die jeweils für die Verarbeitung von Bestellungen, Zahlungen und Versand verantwortlich sind. Jeder Service hat seine eigene Datenbank und REST-Schnittstelle, über die er mit anderen Services und dem Gateway kommuniziert. Änderungen am Bezahl- oder Versandprozess können so unabhängig vom eigentlichen Bestellprozess entwickelt und ausgeliefert werden. Auch besteht die Möglichkeit bei besonders hohen Lasten oder Engpässen, die Services unabhängig voneinander horizontal zu skalieren.

[5] https://github.com/Beleg-6-EAP/demo-soa-ecommerce

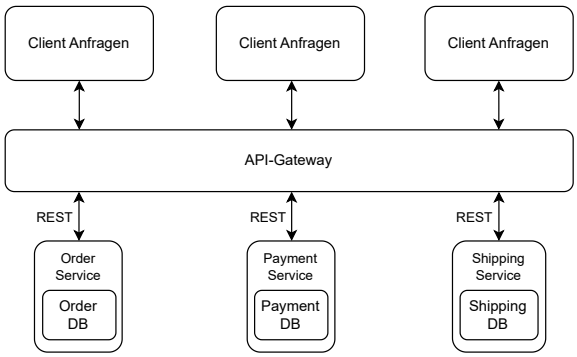

Abb. 7. Aufbau der E-Commerce Anwendung mit Microservices Architecture

Die Microservices Architecture löst viele der Probleme, die in klassischen monolithischen oder Service-oriented Architecture auftreten. Durch die Aufteilung der Anwendung in kleinere, separat ausgelieferte Services, entstehen inhärent robustere und besser skalierbare Systeme. Da Änderungen oft nur einzelne Service Komponenten betreffen, muss nicht das ganze System neu ausgeliefert werden. Es reicht stattdessen, nur den betroffenen Service zu aktualisieren, was die Ausfallzeiten minimiert und die Wartbarkeit der Software erhöht. Vor allem in agilen Umgebungen, in denen viele Iterationen und schnelle Auslieferungen notwendig sind, ist die Microservices Architektur daher besonders geeignet [Richards 2015, 33].

Auch können Services in produktiven Umgebungen ohne Ausfallzeiten ausgetauscht werden. Solange, wie der neue Service ausgeliefert wird, leitet das API-Gateway Anfragen an den alten Service weiter. Erst wenn der neue Service bereit ist, werden Anfragen an diesen weitergeleitet und der alte Service kann abgeschaltet werden.

Microservices sind außerdem sehr gut testbar. Tests können durch die strikte Trennung der Funktionalitäten isoliert auf einzelne Services angewendet werden, ohne dass andere Services betrachtet werden müssen. Auch Regressionstests nehmen dadurch weniger Zeit in Anspruch. Kleine Änderungen haben nicht zur Folge, dass die gesamte Anwendung neu getestet werden muss.

Obwohl die Auslieferungen schnell sind und parallel an verschiedenen Services gearbeitet werden kann, ist die initiale Entwicklungszeit einer Anwendung in Microservices Architecture oft groß. Im Vergleich zu monolithischen Anwendungen ist vor allem mit höherem Zeitaufwand durch Entwurf der Schnittstellen, Absprache unter den Entwicklungsteams und Planung der Architektur zu rechnen [Salah et al. 2016, 6-7].

Der größte Nachteil der sich hier ergibt, ist die vergleichsweise niedrige Performance aufgrund der verteilten Natur des Systems. Die Kommunikation zwischen Services findet meist über das Netzwerk statt, was zu erhöhten Latenzen führen kann[Richards 2015, 34].

6 LAYERED-ARCHITECTURE

Das Problem der redundanten Schnittstellenlogik in Microservices lässt sich durch die Anwendung der Layered Architecture effektiv lösen. Dieses Architekturmuster basiert auf der Grundidee, eine Anwendung in verschiedene Schichten (englisch *layers*) zu gliedern. Dabei bleiben sowohl die Anzahl der Schichten als auch deren spezifischen Aufgaben flexibel und werden nicht durch das Muster vorgeschrieben [Tu 2023, 34].

Diese Flexibilität erlaubt es Unternehmen, die Schichtstruktur individuell an ihre spezifischen Anforderungen anzupassen. Um jedoch Vorteile wie eine verbesserte Wartbarkeit, Skalierbarkeit und Wiederverwendbarkeit zu realisieren, ist die Einhaltung zentraler Prinzipien der Layered Architecture essenziell [Tu 2023, 34].

Ein grundlegendes Prinzip ist die Trennung der Zuständigkeiten (englisch *Separation of Concerns*). Hierbei werden Komponenten mit unterschiedlichen Aufgaben in separate Schichten aufgeteilt, sodass jede Schicht ausschließlich für eine klar definierte und abgeschlossene Funktionalität verantwortlich ist [Tu 2023, 34].

Ein weiteres wesentliches Prinzip ist die Isolation der Schichten (englisch *Layers of Isolation*), welches gewährleistet, dass Änderungen innerhalb einer Schicht lediglich deren eigene Komponenten betreffen und keine Auswirkungen auf andere Schichten haben [Richards 2015, 3–4].

Die Layered Architecture bietet insbesondere für Unternehmen die Möglichkeit, häufig genutzte Funktionalitäten, wie etwa Authentifizierung oder Logging, in dedizierten Schichten zu kapseln und diese flexibel in verschiedenen Diensten wiederzuverwenden. Dabei sorgt eine klar definierte Schnittstelle jeder Schicht sowohl für eine reibungslose Kommunikation als auch für die Abstraktion der internen Implementierung.

Zusätzlich lässt sich die Layered Architecture nahtlos mit modernen agilen Architekturmustern wie Microservices kombinieren, wodurch die Agilität eines Unternehmens weiter gesteigert werden kann.

Ein praxisnahes Beispiel hierfür ist das E-Commerce-Beispiel mit Microservices. In der ursprünglichen Implementierung enthielt jeder Service ein eigenes Modul für die Authentifizierung. Diese redundante Struktur führte jedoch zu erheblichen Herausforderungen hinsichtlich Wartbarkeit und Kosten: Änderungen an der Authentifizierungslogik mussten in jedem einzelnen Service separat durchgeführt werden, was den Entwicklungsprozess verlangsamte und unnötig komplizierte.

Eine Lösung für dieses Problem besteht darin, die Authentifizierungslogik in eine zentrale Schicht auszulagern. Diese Schicht kann direkt mit dem API-Gateway kommunizieren und somit eine einheitliche Authentifizierung gewährleisten (siehe Abb. 8).

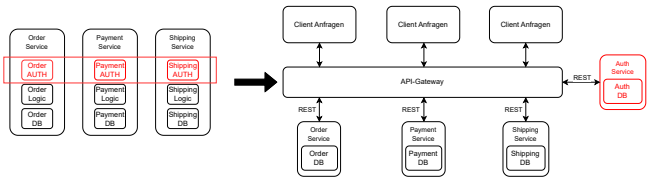

Abb. 8. Layered Microservice Architecture: Beispiel E-Commerce

Mit der Einführung einer zentralen Authentifizierungsschicht kann die Authentifizierungslogik an einer einzigen Stelle gebündelt und bei Bedarf angepasst werden, ohne Änderungen in den einzelnen Services vornehmen zu müssen. Dies bietet insbesondere in Unternehmen mit einer Vielzahl von Services signifikante Zeit- und Kosteneinsparungen.

Ein weiterer Vorteil besteht darin, dass die Authentifizierung für sämtliche Dienste zentral über ein API-Gateway durchgeführt wird, wodurch eine wiederholte Authentifizierung auf Service-Ebene entfällt. Diese Entlastung erlaubt es den einzelnen Services, sich ausschließlich auf ihre Kernfunktionalität zu konzentrieren, während die Authentifizierung ausgelagert ist.

Die vollständige Implementierung des E-Commerce-Beispiels mit geschichteter Microservice-Architektur ist bei GitHub [6] zu finden.

Die klare Trennung der Verantwortlichkeiten fördert die Zusammenarbeit in kleinen, autonomen Teams, da jedes Team unabhängig an einer Schicht arbeiten kann. Zudem ermöglicht das Prinzip der Isolation der Schichten eine höhere Flexibilität gegenüber wechselnden Anforderungen. Durch die Wiederverwendbarkeit in der Layered Architecture wird Mehraufwand vermieden und duplizierter Code reduziert, was zu kürzeren Iterationen und häufigeren Auslieferungen führt.

Das Beispiel verdeutlicht zudem, dass die Layered Architecture sich hervorragend mit agilen Architekturen wie Microservices kombinieren lässt. Diese Synergie trägt entscheidend dazu bei, die Agilität des Systems nachhaltig zu steigern.

7 EVENT-DRIVEN ARCHITECTURE

Die Event-Driven Architecture wählt als Basis einen anderen Ausgangspunkt als die bisherigen Architekturmuster. Während bei letzteren Komponenten Dienste bereitstellen, welche von anderen Komponenten explizit genutzt werden, verhalten sich Dienstbereitstellende Komponenten in der Event-Driven Architecture reaktiv, werden also implizit von Dienst-konsumierenden Komponenten genutzt [Garlan and Shaw 1994]. Ein System reagiert somit asynchron auf Zustandsänderungen, also Ereignisse in diesem System [Manchana 2021]. Die in dieser Architektur minimalen Einheiten, welche Informationen einer Zustandsänderung kapseln, werden *Events* genannt. Die Idee der impliziten Behandlung von Ereignissen ist nicht neu und taucht erstmals 1994 im von Garlan und Shaw publizierten Papier *„An introduction to Software Architecture“* auf.

Betrachten wir im Folgenden die Basis-Bestandteile der Event-Driven Architecture:

- Ereignis (englisch *Event*): Kapselt Information einer Zustandsänderung eines Systems
- Produzent (englisch *Producer*): Komponente, die Event erzeugt
- Herausgeber (englisch *Publisher*): Komponente, die, von Produzenten erzeugte, Events publiziert
- Konsument (englisch *Consumer*): Komponente, die auf publizierte Events reagiert
- Vermittler (englisch *Mediator*): Komponente zwischen Produzenten und Konsumenten - filtert Events und verteilt diese auf Konsumenten
- Event-Bus: Oft auch *Event-Broker* genannt - bietet die Infrastruktur für die Gesamtheit der Vermittler

Abstrakt kann ein Event als Vertrag zwischen Produzenten und Konsumenten am Event-Bus betrachtet werden. Der Konsument nutzt die Spezifikation des Events am Bus, der Produzent implementiert jene Spezifikation. Abbildung 9 stellt diesen Vertrag dar.

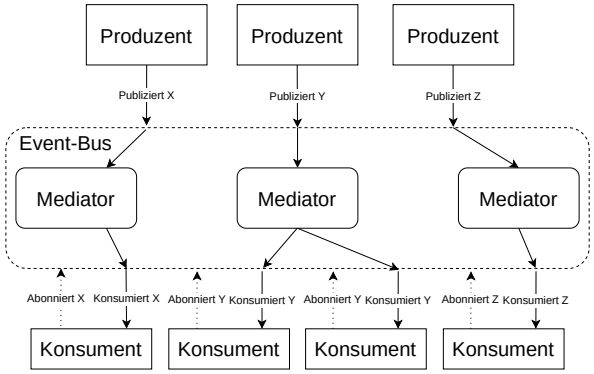

Abb. 9. Vertrag zwischen Produzenten und Konsumenten am Event-Bus

Durch den Vertrag weisen die Events am Event-Bus starke Kohäsion und somit lose Kopplung auf. Diese lose Kopplung minimiert nicht nur kaskadierende Fehler, sondern ermöglicht kleinen und autonomen Entwickler-Teams die klare Abgrenzung von Features und folglich einfach definierbare Iterationen - eine Menge von Events, deren Erzeugung und Konsumierung.

Weiter sind Events oft nah an dem, was Ereignisse in realen Prozessen sind, also domain-driven. Gebündelt ermöglichen obige Punkte die kontinuierliche Auslieferung von Software in kurzen Intervallen.

Außerdem garantiert die asynchrone Behandlung von Ereignissen zusammen mit der loosen Kopplung hohe Skalierung und die Möglichkeit für Echtzeit-Software. Daher sind Event-Driven Architekturen besonders für datenintensive Echtzeit-Anwendungen wie IoT (Internet of Things) und Analytics geeignet [Siddiqui et al. 2023].

Betrachten wir erneut das E-Commerce-Beispiel aus der Einleitung. Dafür definieren wir drei Arten von Events:

- `OrderCreated`: Ein Event, das genau dann erzeugt wird, wenn eine neue Bestellung aufgegeben wird
- `PaymentProcessed`: Ein Event, das genau dann erzeugt wird, wenn der Bezahlvorgang abgeschlossen wird
- `ShipmentInitiated`: Ein Event, das genau dann erzeugt wird, wenn die Bestellung versandt wird

Analog zur Microservice-Architektur teilen wir die Funktionalitäten in drei verschiedene Dienste auf: `OrderService`, `PaymentService` und `ShipmentService`.

[6] https://github.com/Beleg-6-EAP/demo-microservice-ecommerce

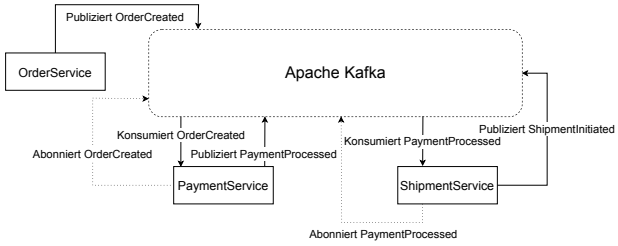

Abb. 10. E-Commerce-Beispiel mit Event-Driven Architecture

Wie Abbildung 10 zeigt, sind alle drei Dienste Produzenten und Publisher, erzeugen also Events und veröffentlichen diese. Die Dienste `PaymentService` und `ShipmentService` sind zudem Konsumenten, sodass ersterer auf Events des Typs `OrderCreated` und zweiterer auf Events des Typs `ShipmentInitiated` reagiert. Eine beispielhafte Implementierung des `PaymentService` mit Apache Kafka als Event-Broker ist im Anhang A.5 zu finden. Die vollständige Implementierung des E-Commerce-Beispiels ist bei GitHub [7] zu finden.

Das Beispiel zeigt, dass die Event-Driven Architektur mit weiteren agilen Strukturen wie Microservices kombiniert werden kann, was die Agilität der Architektur weiter erhöht. Die damit einhergehende Komplexität stellt teilweise hohe Anforderungen an die Entwickler. Aufgrund der Asynchronität der Behandlung von Ereignissen ist die Testung des Systems meist schwer und die Fehlerbehandlung essentiell. Mögliche Problemquellen schließen dabei unter anderem Event-Verlust, erhöhte Latenz und Inkonsistenz ein. Die hohen Anforderungen an die Entwickler verlangen viel Vertrauen in jene, einer der zentralen Punkte des agilen Manifests [Michl 2018]. Insgesamt weist die Event-Driven Architecture also eine sehr hohe Agilität auf und ist damit besonders für moderne Software und ihre stetig wechselnden Anforderungen geeignet.

8 CLOUD-NATIVE ARCHITECTURE

Die Cloud-Native Architecture beruht auf der Annahme, dass die Infrastruktur in ständigem Wandel ist und die Auslagerung jener in die Cloud mehr Agilität schafft [Gannon et al. 2017]. Als Cloud wird in diesem Fall die Infrastruktur eines oder mehrerer Cloud-Vendors bezeichnet. Ein Cloud-Vendor bietet seine Infrastruktur und deren Verwaltung dabei gegen eine Gebühr an. Unter anderem offeriert er:

- Die globale Nutzung von Ressourcen durch Geo-Redundanz und somit starke Verteilung sowie hohe Verfügbarkeit von Software,
- dynamische Skalierung bereitgestellter Ressourcen basierend auf der Nachfrage von Software (Auto-Scaling),
- nutzungsbedingte Gebühren, sodass nur tatsächlich verwendete Ressourcen bezahlt werden (Pay-as-you-go),
- unterbrechungsfreie Updates von Software (Zero Downtime).

Als cloud-nativ wird hierbei all jene Software bezeichnet, welche explizit für die Cloud entwickelt wurde. Grundsätzlich baut die cloud-native Architektur auf die in diesem Artikel bereits in Abschnitt **??** erklärte Microservice-Architektur und die in Abschnitt 7 erklärte Event-Driven Architecture auf. Zusätzlich kommen sogenannte *Fully Managed Cloud-Services* hinzu, was cloud-spezifische und vom Cloud-Vendor vollständig verwaltete Dienste sind. Jene umfassen alle von Datenbanken über Message-Queues bis hin zu Serverless-Functions und viele mehr. Charakterisiert werden diese durch die Eigenschaft, dass sich der Entwickler gänzlich auf die Business-Logik konzentrieren kann, da der Cloud-Provider die vollständige Verwaltung der Infrastruktur übernimmt. Zentral dabei ist der Aspekt der Containerisierung, bei welchem jede Komponente eines Systems in einen Container gepackt wird. Der Cloud-Vendor kann die Gesamtheit der Container dann dynamisch orchestrieren und somit gezielt den Ressourcenverbrauch optimieren.

Im Folgenden greifen wir erneut das Beispiel E-Commerce aus der Einleitung auf und betrachten die Anpassung der in Abschnitt 7 angewendeten Event-Driven-Architecture auf die Cloud-Native Architecture mit Cloud-Vendor AWS.

Dabei ersetzen wir den dort verwendeten Broker durch eine *EventBridge* [8] - ein serverless Cloud-Service von AWS zum Routen von Ereignissen. Weiter werden die drei Services für Aufgeben einer Bestellung, Bearbeitung der Bezahlung und Initiierung des Versands als *Lambda* [9], also Serverless-Function implementiert. Abbildung 11 stellt diese cloud-native Architektur dar. Eine beispielhafte Implementierung des `ProcessPaymentLambdas` ist im Anhang A.6 und auf GitHub[10] zu finden.

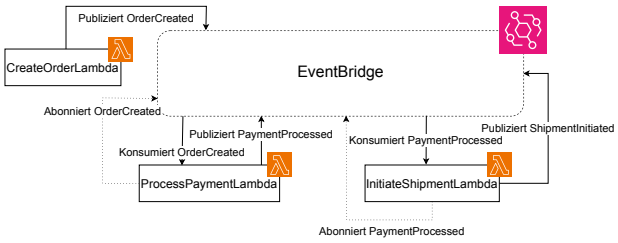

Abb. 11. E-Commerce-Beispiel mit Cloud-Native Architecture

Damit kombiniert die Cloud-Native Architecture die agilen Vorteile der Microservice- und Event-Driven Architecture. Weiter ermöglichen die Cloud-Vendors nahtlose und einfache Möglichkeiten zur Auslieferung der Software, wie beispielsweise Code-Deploy [11] bei AWS. Die Option, allen Fokus von der Infrastruktur auf die Entwicklung zu legen, ermöglicht gemeinsam mit der einfachen Auslieferung kurze Iterationen in agilen Teams und somit noch kürzere Zyklen in der Bereitstellung von Software. Dabei ist besonders die Time-to-Market sehr gering, sodass Tech-Start-Ups häufig cloud-native Architekturen für ihre Software wählen.

Nicht zu vergessen ist zudem die extrem hohe Flexibilität der eingesetzten Ressourcen durch das Auto-Scaling. Das damit verbundene Pay-as-you-go ermöglicht weiter auch finanzielle Agilität, wodurch zum Beispiel keine Vorab-Investitionen für Infrastruktur notwendig sind. Jedoch ist zu betonen, dass aufgrund dessen

[7] https://github.com/Beleg-6-EAP/demo-eda-ecommerce

[8] https://aws.amazon.com/de/eventbridge/, abgerufen am 06.01.2025
[9] https://aws.amazon.com/de/lambda/, abgerufen am 06.01.2025
[10] https://github.com/Beleg-6-EAP/demo-cloud-native-ecommerce
[11] https://aws.amazon.com/de/codedeploy/, abgerufen am 06.01.2024

auch Kostenrisiken durch möglicherweise unerwartete, extrem hohe Nachfrage der Software bestehen. Ebenfalls nicht zu vernachlässigen ist die enge Bindung an den Cloud-Vendor durch Verwendung seiner spezifischen Cloud-Dienste. Im Fall einer Migration zu einem anderen Cloud-Vendor könnten sowohl hohe Kosten beim vorherigen Cloud-Vendor, als auch Entwicklungskosten durch die notwendige Anpassung der genutzten spezifischen Cloud-Dienste anfallen. Obwohl das die Wahl des Cloud-Anbieters und der damit verbunden Entwicklung weniger agil macht, gilt die Cloud-Native Architecture besonders aufgrund der Auslagerung der Infrastruktur als eine - und vermutlich die agilste Architektur für moderne Software.

9 MICROKERNEL ARCHITECTURE

Die Microkernel Architecture ist ein Architekturmuster, was sich durch Erweiterbarkeit, Flexibilität und vor allem Isolation der Funktionalitäten auszeichnet. Wie in Abbildung 12 dargestellt, enthält ein Microkernel zwei wesentliche Komponenten: Den Kern der Anwendung,der die wichtigsten grundlegenden Funktionalitäten bereitstellt und Module oder auch Plugins, die diesen Kern um Features erweitern [Richards 2015, 21-22].

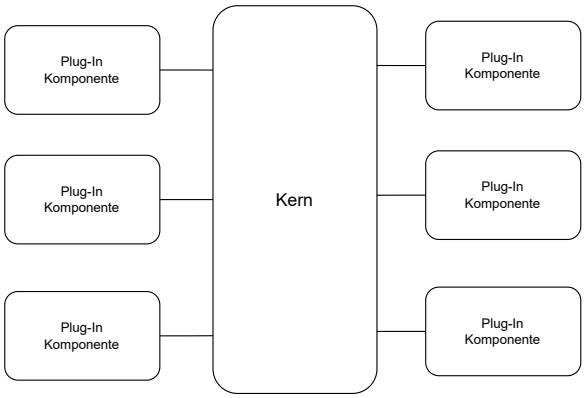

Abb. 12. Aufbau einer Microkernel Architecture

Der Kern der Anwendung implementiert dabei meist nur die minimalste Funktionalität, um die Anwendung oder das System lauffähig zu machen. Alle weiteren Funktionalitäten werden in Modulen implementiert, die auf den Kern aufbauen. Module sind meist unabhängig voneinander aufgebaut, es kann jedoch auch vorkommen, dass manche Module von anderen abhängig sind. Best practice für die Entwicklung von Microkernel Architekturen ist es, die Kommunikation zwischen einzelnen Modulen so gering wie möglich zu halten, um Probleme durch Abhängigkeiten zu vermeiden. Dadurch sind Module untereinander lose gekoppelt und können unabhängig voneinander entwickelt, getestet und deployed werden [Richards 2015, 22].

Die Plugins können über verschiedene Wege mit dem Kern verbunden werden. Eine genaue Spezifikation zum Verbinden der Plugins mit dem Kern gibt es aber laut Architekturschema nicht, diese Entscheidung ist dem Entwickler überlassen und entsprechend der Anforderungen und Anwendungsumgebung zu treffen. Unabhängig von der Art der Verbindung definiert der Kern die Schnittstellen, um Plugins anzubinden. Diese Verbindung könnte dann beispielsweise über Web-Services, Messaging oder am einfachsten über direkte Objekt-Instanziierung innerhalb der gleichen Anwendung stattfinden [Richards 2015, 22-23].

Zwischen Plugins und Kern werden Verträge definiert, die die Kommunikation zwischen den beiden Komponenten regeln. Diese Verträge können in Form von Interfaces, Klassen oder auch Datenstrukturen definiert werden. Alle Plugins, müssen sich zwingend an die definierten Verträge halten, um mit dem Kern kommunizieren zu können. Alternativ können auch Adapter verwendet werden, um bestehende Plugins an den Kern und die Verträge anzupassen, wodurch wiederrum die lose Kopplung der Komponenten verbessert wird.

Durch diesen Aufbau ergibt sich jedoch das Problem, dass der Kern jederzeit über Verfügbarkeit und Erreichbarkeit der Plugins informiert sein muss. Um dieses Problem zu lösen, kann eine zentrale Plugin-Registry verwendet werden. Diese Registry enthält alle aktuell verfügbaren Plugins sowie die dazugehörigen relevanten Informationen wie zum Beispiel Name des Service, Verträge, Verbindungsdetails, etc. Der Kern der Anwendung kann dann zur Laufzeit auf diese Registry zugreifen und Plugins dynamisch laden [Richards 2015, 22].

Microkernel Architekturen können auch in andere Architekturmuster eingebettet werden, falls es nicht möglich sein sollte die gesamte Software in diesem Architekturmuster aufzubauen. Vor allem Teile von Anwendungen, die stark erweiterbar sein müssen, eignen sich gut für die Verwendung der Microkernel Architektur.

Ein klarer Vorteil dieses Architekturmusters ist die schnelle Reaktionsfähigkeit auf äußere Änderungen, da Anpassungen aufgrund der losen Kopplung größtenteils nur in den isolierten Modulen vorgenommen werden. Der Kern der Anwendung ist in den meisten Fällen schnell stabil und benötigt selten im Laufe der Entwicklung weitere Angleichungen. Geänderte Module können je nach Implementierung auch zur Laufzeit geladen oder hinzugefügt werden, was mögliche Downtime von bereits ausgelieferter Software minimiert [Richards 2015, 25].

Ein Beispiel dafür stellt die Entwicklung von Betriebssystem-Kernels dar, die auch namensgebend für dieses Architekturmuster ist. Deren Kern Komponenten sind in der Regel sehr stabil und implementieren vor allem grundlegende Funktionen wie Speicherverwaltung, Prozessverwaltung und I/O-Operationen. Weitere low-level Funktionalitäten wie beispielsweise Geräte Treiber oder Dateisysteme werden als Module in den Kernel geladen und können bei Bedarf hinzugefügt oder entfernt werden, was vor allem für die Unterstützung neuer Hardware wichtig ist.

Auch Entwicklungsumgebungen in der Softwareentwicklung nutzen oft Microkernel Architekturen, um die Support für verschiedene Programmiersprachen und Frameworks zu ermöglichen.

Zwar bietet das Beispiel der E-Commerce Anwendung keinen klassischen Anwendungsfall für Microkernel Architekturen, jedoch kann die Verwendung von Microkernel Architekturen in Teilen der Anwendung trotzdem sinnvoll sein. Sowohl Zahlungs- als auch Versandfunktionalitäten könnten, wie in Abbildung 13 dargestellt, in Module ausgelagert werden, um die Anwendung durch weitere Dienstleister erweitern zu können.

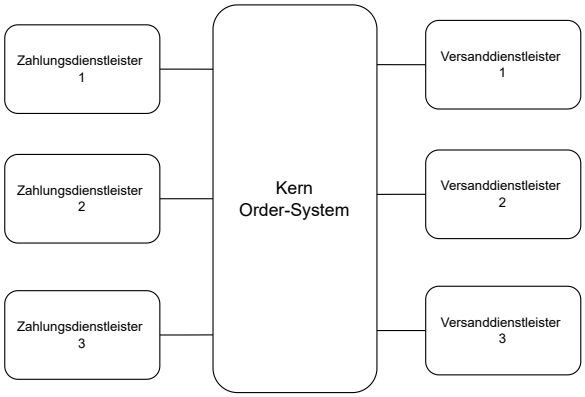

Abb. 13. Aufbau der E-Commerce Anwendung mit Microkernel Architektur

Logik zum Verarbeiten der Zahlungen und Versandinformationen wird dann in den Modulen implementiert, die auf den Kern der Anwendung aufbauen. Dabei ist jedoch ein Großteil der Business-Logik im Kern der Anwendung enthalten, was nicht dem eigentlichen Gedanken der Microkernel Architektur entspricht.

Abgesehen davon erhöhen Microkernel Architekturen inhärent die Testbarkeit der Software, da Module nur lose Kopplung gekoppelt sind. Jedes Modul kann unabhängig voneinander getestet werden und fehlende Module durch Stubs ersetzt werden, wodurch sich während der Entwicklung auf einzelne Module isoliert konzentriert werden kann [Richards 2015, 26]. Weiterhin können Verhaltensweisen von anderen Modulen durch Mocks simuliert werden, um Testzustände zu erzeugen und das Verhalten der Anwendung zu verifizieren. In Agilen Umgebungen, in denen das Testen von Software eine wichtige Rolle spielt, ist die Verwendung von Microkernel Architekturen daher besonders sinnvoll.

Eine Herausforderung bei der Verwendung von Microkernel Architekturen kann jedoch der Entwurf der Kern-Komponente darstellen. Da alle anderen Module auf den Kern aufbauen, muss dieser sehr sorgfältig und stabil entwickelt werden, um die Funktionalität der gesamten Anwendung zu gewährleisten. Diese Rolle sollten vor allem erfahrene Entwickler übernehmen, da sich Design-Fehler der Kern-Komponente oder Verträge negativ auf die Entwicklung der Module auswirken können. Sollte der Kern der Anwendung angepasst werden, so müssen tendenziell auch alle Module überprüft oder aktualisiert werden, was zu erheblich erhöhten Entwicklungszeiten führen kann und zuvor gewonnene Vorteile der Microkernel Architektur zunichte macht.

Aufgrund der initial hohen Komplexität, die mit der Entwicklung des Kerns einhergeht, stellt die Microkernel Architektur nicht die beste Wahl dar, wenn es darum geht schnell eine erste Version der Software auf den Markt zu bekommen. Die wesentlichen Vorteile, die die Microkernel Architektur bietet, zeigen sich erst im späteren Verlauf der Entwicklung, wenn die Anwendung erweitert werden muss. Sowohl Iterationen als auch Auslieferungszeiten sind dann sehr kurz, da Module unabhängig voneinander entwickelt werden und die Anwendung schnell an neue Anforderungen angepasst werden kann. Diese Vorteile können die Architektur in ausgewählten Anwendungsfällen sehr geeignet für agile Entwicklungsumgebungen machen.

10 ZUSAMMENFASSUNG UND AUSBLICK

In dieser Arbeit wurden verschiedene Enterprise-Architektur-Muster untersucht und deren Eignung für moderne Geschäftsprozesse bewertet. Dabei wurde festgestellt, dass die Agilität im Entwicklungsprozess maßgeblich durch die Wahl des Architekturmusters beeinflusst wird.

Die monolithische Architektur zeichnet sich durch die Verwendung einer einzigen Komponente aus, die alle Funktionalitäten der Anwendung bereitstellt. Modular monolithische Architekturen bauen auf dieser Idee auf, indem sie die monolithische Anwendung in Module unterteilen, um die Wartbarkeit zu verbessern. SOA teilt die Anwendung in grobe Dienste auf, was die Wiederverwendung von Funktionalitäten ermöglicht. Microservices verbessern die Skalierbarkeit und Wartbarkeit von SOA, indem sie die Dienste in noch kleinere, isolierte Komponenten aufteilen, die nur über standardisierte Schnittstellen kommunizieren. Das Alleinstellungsmerkmal von Event-Driven Architekturen sind die asynchronen Reaktionen auf Ereignisse, wodurch die Anwendung sehr flexibel und skalierbar wird. Cloud-native Architekturen lagern die Verwaltung der Infrastruktur an Cloud-Anbieter aus, sodass sich in der Entwicklung auf die Business-Logik konzentriert werden kann. Als Architektur für besonders erweiterbare Anwendungen lässt sich die Microkernel-Architektur verwenden, die die Anwendung auf einen Kern mit wenig Funktionalität und Plugins mit Business-Logik auftrennt.

Eine Bewertung der Architekturen anhand von Agilen Kriterien ist in Tabelle 1 dargestellt.

	Monolith	Mod. Monol.	SOA	Microservices	Layered	EDA	Cloud-Native	Microkernel
Time-To-Market	✓✓✓	✓✓✓	0	✗	?	✗	✓✓	✗✗✗
Skalierbarkeit	✗✗✗	✗✗✗	✗	✓✓✓	?	✓✓✓	✓✓✓	0
Kostenflexibilität	✗✗	✗✗	✗	✓	?	✓	✓✓✓	✗
Anforderungsänderungen	✗✗✗	✗✗	✓	✓✓✓	✓✓	✓✓✓	✓✓✓	✓✓
Erweiterbarkeit	✗✗✗	✗✗	0	✓✓	✓✓	✓✓	✓✓	✓✓✓
Auslieferung	✗✗✗	✗✗✗	✓	✓✓	✓	✓✓	✓✓✓	✓✓

Tabelle 1. Vergleich der Architekturmuster

Auffallend ist zum einen, dass monolithische Anwendungen zwar die beste Time-To-Market haben, den anderen Architekturmustern jedoch in allen anderen Vergleichspunkten unterlegen. Die Architekturen mit der besten Skalierbarkeit zeichnen sich durch die starke Isolation von Komponenten aus und sind hier Microservices, Event-Driven und Cloud-Native Architekturen. Die einzige Architektur, die eine hohe Flexibilität bei den Kosten aufweist, ist die Cloud-Native Architektur, da nur die tatsächlich genutzten Ressourcen bezahlt werden. Auf Änderungen in den Anforderungen können alle modernen Architekturen sehr gut reagieren. Die beste Erweiterbarkeit weist die Microkernel Architektur auf, da die Business-Logik größtenteils durch Plugins realisiert wird. Betrachten wir die Auslieferung von Software, so schneiden traditionelle Architekturen

hier am schlechtesten ab, da sie nur im Ganzen ausgetauscht werden können. Moderne Architekturen, insbesondere Microservices, Event-Driven und Cloud-Native Architekturen, können Software in kleinen Teilen ausliefern und so schneller auf Änderungen der Anforderungen reagieren. Die Layered Architektur ist hier teilweise schwer einzuordnen, da sie meistens mit anderen Architekturmustern kombiniert wird, die einen wesentlichen Einfluss auf Time-To-Market, Skalierbarkeit und Kosten haben.

Betrachten wir das E-Commerce-Beispiel und die anfangs gestellten Anforderungen, so zeigt sich, dass die Cloud-Native Architektur in diesem Fall die beste Wahl darstellt. Es kann schnell auf technische Änderungen reagiert werden, die in produktiv-Umgebungen einfach ausgerollt werden können. Wachsender Traffic, der später auftreten könnte, kann durch Auto-Scaling einfach bewältigt werden. Entwickler, die sich auf den Aufbau der Infrastruktur konzentrieren, sind nicht nötig und können sich an der Entwicklung der Business-Logik beteiligen. Die Kosten sind flexibel und nur abhängig von der tatsächlichen Nutzung der Ressourcen, was vor allem am Anfang bei geringen Nutzerzahlen Kosten spart. Aufgrund der geringen Time-To-Market ist es außerdem möglich, einen MVP schnell auf den Markt zu bringen und diesen dann iterativ zu verbessern.

Unabhängig davon gilt jedoch, dass es kein universell bestes Architekturmuster gibt. Die Wahl der Architektur muss den spezifischen Anforderungen eines Projekts angepasst sein und besteht meist aus einer Kombination mehrerer EA-Muster.

Damit bleibt die Frage offen, wie hybride Architekturen aussehen könnten, die Elemente aus verschiedenen EA-Architekturen kombinieren, um alle agilen Anforderung bestmöglich zu erfüllen.

LITERATUR

Kalpesh Barde. 2023. Modular Monoliths: Revolutionizing Software Architecture for Efficient Payment Systems in Fintech. (2023). https://doi.org/10.1145/3643657.3643911

Phil Bianco, Rick Kotermanski, and Paulo Merson. 2007. *Evaluating a service-oriented architecture.* Citeseer.

Grzegorz Blinowski, Anna Ojdowska, and Adam Przybyłek. 2022. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access* 10 (2022), 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803

Rui Chen, Shanshan Li, and Zheng Li. 2017. From Monolith to Microservices: A Dataflow-Driven Approach. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC).* 466–475. https://doi.org/10.1109/APSEC.2017.53

Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pål Krogdahl, Min Luo, and Tony Newling. 2004. *Patterns: service-oriented architecture and web services.* IBM Corporation, International Technical Support Organization New York, NY ....

Dennis Gannon, Roger Barga, and Neel Sundaresan. 2017. Cloud-Native Applications. *IEEE Cloud Computing* 4, 5 (2017), 16–21. https://doi.org/10.1109/MCC.2017.4250939

David Garlan and Mary Shaw. 1994. *An Introduction to Software Architecture.* Technical Report CMU/SEI-94-TR-021. https://insights.sei.cmu.edu/library/an-introduction-to-software-architecture/ Accessed: 2025-Jan-2.

Danny Greefhorst and Erik Proper. 2011. *The Role of Enterprise Architecture.* Springer Berlin Heidelberg, Berlin, Heidelberg, 7–29. https://doi.org/10.1007/978-3-642-20279-7_2

Guozhi Liu, Bi Huang, Zhihong Liang, Minmin Qin, Hua Zhou, and Zhang Li. 2020. Microservices: architecture, container, and challenges. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C).* 629–635. https://doi.org/10.1109/QRS-C51114.2020.00107

Ramakrishna Manchana. 2021. Event-Driven Architecture: Building Responsive and Scalable Systems for Modern Industries. *International Journal of Science and Research (IJSR)* 10 (01 2021), 1706–1716. https://doi.org/10.21275/SR24820051042

Thomas Michl. 2018. *Das agile Manifest – eine Einführung.* Springer Berlin Heidelberg, Berlin, Heidelberg, 3–13. https://doi.org/10.1007/978-3-662-57699-1_1

Mark Richards. 2015. *Software Architecture Patterns.* O'Reilly Media, Inc. value pages.

Tasneem Salah, M. Jamal Zemerly, Chan Yeob Yeun, Mahmoud Al-Qutayri, and Yousof Al-Hammadi. 2016. The evolution of distributed systems towards microservices architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST).* 318–325. https://doi.org/10.1109/ICITST.2016.7856721

Nada Salaheddin and Nuredin Ahmed. 2022. MICROSERVICES VS. MONOLITHIC ARCHITECTURE [THE DIFFERENTIAL STRUCTURE BETWEEN TWO ARCHITECTURES]. *MINAR International Journal of Applied Sciences and Technology* 4 (10 2022), 484–490. https://doi.org/10.47832/2717-8234.12.47

Hassaan Siddiqui, Ferhat Khendek, and Maria Toeroe. 2023. Microservices based architectures for IoT systems - State-of-the-art review. *Internet of Things* 23 (2023), 100854. https://doi.org/10.1016/j.iot.2023.100854

Ruoyu Su and Xiaozhou Li. 2024. Modular Monolith: Is This the Trend in Software Architecture?. In *Proceedings of the 1st International Workshop on New Trends in Software Architecture* (Lisbon, Portugal) *(SATrends '24).* Association for Computing Machinery, New York, NY, USA, 10–13. https://doi.org/10.1145/3643657.3643911

Zhenan Tu. 2023. Research on the Application of Layered Architecture in Computer Software Development. *Journal of Computing and Electronic Information Management* 11 (11 2023), 34–38. https://doi.org/ojs/index.php/jceim/article/view/14398

A CODE-BEISPIELE

A.1 Monolithic Architecture

```java
package demo.monolith.service;

import demo.monolith.model.Payment;
import demo.monolith.repository.PaymentRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.UUID;

@Service
@RequiredArgsConstructor
public class PaymentService {

    private final PaymentRepository paymentRepository;

    public void processPayment(String orderId) {
        final Payment payment = new Payment(UUID.randomUUID().toString(), orderId, true);
        paymentRepository.save(payment);
    }

    public List<Payment> getAll() {
        return paymentRepository.findAll();
    }
}
```

Listing 1. Service-Implementierung des `PaymentService` in Java Spring Boot 3.4.1

A.2 Modular Monolithic Architecture

```java
package demo.modulith.order.internal;

import demo.modulith.order.OrderService;
import demo.modulith.payment.PaymentService;
import demo.modulith.payment.internal.PaymentServiceImpl;
import demo.modulith.shipment.ShipmentService;
import demo.modulith.shipment.internal.ShipmentServiceImpl;
import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
@RequiredArgsConstructor
public class OrderServiceImpl implements OrderService {

    private final OrderRepository orderRepository;
    private final PaymentService paymentService;
    private final ShipmentService shipmentService;

    public Order create(Order order) {
        final Order savedOrder = orderRepository.save(order);
        paymentService.processPayment(savedOrder.getId());
        shipmentService.handleShipping(savedOrder.getId());

        return savedOrder;
    }

    public List<Order> getAll() {
        return orderRepository.findAll();
    }
}
```

Listing 2. Service-Implementierung des `OrderService` in Java Spring Boot 3.4.1

A.3 Microservices Architecture

```kotlin
package demo.microkernel.orderservice.internal

import demo.microkernel.orderservice.client.RestClient
import demo.microkernel.orderservice.internal.model.Order
import demo.microkernel.orderservice.service.OrderService
import org.springframework.stereotype.Service


@Service
internal class OrderServiceImpl(
    private val orderRepository: OrderRepository,
    private val restClient: RestClient,
) : OrderService {

    override fun create(order: Order): Order {
        val savedOrder: Order = orderRepository.save(order)
        restClient.postPayment(savedOrder.id)
        restClient.postShipment(savedOrder.id)
        restClient.postLog("Order created: ${savedOrder.id} for customer: ${savedOrder.userId}")
        return savedOrder
    }

    override fun getAll(): List<Order> =
        orderRepository.findAll()
}
```

Listing 3. Service-Implementierung des `OrderService` in Kotlin Spring Boot 3.4.1

A.4 Layered Architecture

```kotlin
package demo.microkernel.loggingservice.internal

import demo.microkernel.loggingservice.service.LoggingService
import org.slf4j.Logger
import org.slf4j.LoggerFactory
import org.springframework.stereotype.Service

@Service
class LoggingServiceImpl: LoggingService {

    val logger: Logger = LoggerFactory.getLogger(LoggingServiceImpl::class.java)

    override fun logMessage(serviceName: String, message: String) {
        logger.info("[$serviceName] - $message")
    }
}
```

Listing 4. Service-Implementierung des `LoggingService` in Kotlin Spring Boot 3.4.1

A.5 Event-Driven Architecture

```java
package demo.eda.service;

import demo.eda.event.OrderCreatedEvent;
import demo.eda.event.PaymentProcessedEvent;
import demo.eda.model.Payment;
import demo.eda.repository.PaymentRepository;
import lombok.RequiredArgsConstructor;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.kafka.core.reactive.ReactiveKafkaProducerTemplate;
import org.springframework.stereotype.Service;
import reactor.core.publisher.Mono;

import java.util.UUID;

@Service
@RequiredArgsConstructor
public class PaymentService {

    private final PaymentRepository paymentRepository;
    private final ReactiveKafkaProducerTemplate<String, PaymentProcessedEvent> producer;

    @KafkaListener(topics = OrderCreatedEvent.TOPIC, groupId = "payment-service")
    public Mono<Void> processPayment(OrderCreatedEvent event) {
        final Payment payment = new Payment(UUID.randomUUID().toString(), event.getOrderId(), true);
        return paymentRepository.save(payment).flatMap(savedPayment -> {
            final PaymentProcessedEvent paymentEvent =
                    new PaymentProcessedEvent(
                            savedPayment.getOrderId(),
                            savedPayment.getId(),
                            savedPayment.isSuccess());
            return producer.send(PaymentProcessedEvent.TOPIC, paymentEvent).then();
        }).then();
    }
}
```

Listing 5. Service-Implementierung des `PaymentService` in Java Spring Boot 3.4.1 mit Apache Kafka als Event-Broker

A.6 Cloud-native Architecture

```haskell
{-# LANGUAGE GHC2024 #-}
{-# LANGUAGE DeriveAnyClass #-}

module PaymentService where

import AWS.Lambda.Runtime (ioRuntime)
import Data.Aeson
import Data.UUID.V4 (nextRandom)
import Data.UUID (toString)
import GHC.Generics (Generic)

data OrderCreatedEvent = OrderCreatedEvent
  { orderId :: String
  , userId  :: String
  , amount  :: Double
  } deriving stock (Show, Generic)
  deriving anyclass (FromJSON, ToJSON)

data Payment = Payment
  { id             :: String
  , paymentOrderId :: String
  , success        :: Bool
  } deriving stock (Show, Generic)
  deriving anyclass (FromJSON, ToJSON)

orderCreatedEventHandler :: OrderCreatedEvent -> IO ()
orderCreatedEventHandler event = do
  uId <- nextRandom
  let payment = Payment { id = toString uId, paymentOrderId = orderId event, success = True }
  -- imagine here that payment always succeeds and we serialize it successfully
  -- imagine further serialization into e.g. Amazon RDS which would trigger a PaymentProcessedEvent
  return ()

-- entry point for aws lambda
main :: IO ()
main = ioRuntime $ fmap Right . orderCreatedEventHandler
```

Listing 6. Implementierung des `ProcessPaymentLambda`s in Haskell

A.7 Service-oriented Architecture

```ruby
class ServiceRegistry
  def initialize
    @services = []
  end

  def self.instance
    @instance ||= ServiceRegistry.new
  end

  def register_service(name, description, endpoint)
    @services << { name: name, description: description, endpoint: endpoint }
  end

  def find_service(name)
    @services.find { |service| service[:name] == name }
  end
end
```

Listing 7. Implementierung der `ServiceRegistry` in Ruby on Rails

B ÜBUNGSAUFGABEN

B.1 Übungsaufgabe 1

Das E-Commerce-Beispiel aus der Einleitung soll um Nutzer-Authentifizierung erweitert werden. Sie haben sich zuvor für eine Microservice-Architektur entschieden und die in der Einleitung genannten Anforderungen implementiert. Die Authentifizierung wird in verschiedenen Komponenten benötigt. Erläutern Sie, wie Sie die Authentifizierung in die Architektur integrieren.

B.2 Übungsaufgabe 2

Das E-Commerce-Beispiel aus der Einleitung soll um Logging erweitert werden. Sie haben sich zuvor für eine cloud-native Event-Driven-Architektur entschieden und die in der Einleitung genannten Anforderungen implementiert. Das Logging wird in verschiedenen Komponenten benötigt. Erläutern Sie, wie Sie das Logging in die Architektur integrieren. Bedenken Sie dabei, dass Logs aus verschiedenen Komponenten möglicherweise zur Auswertung zusammengeführt werden müssen und somit die Reihenfolge von Logs relevant ist.

B.3 Übungsaufgabe 3

Das E-Commerce-Beispiel aus der Einleitung soll um ein E-Mail-Notifikationssystem erweitert werden. Dieses soll Nutzern E-Mails bei jeder Statusänderung einer ihrer Bestellungen zustellen. Untersuchen Sie für alle in diesem Papier betrachteten Architekturen, wie das E-Mail-Notifikationssystem in die bestehende Architektur integriert werden kann und welche Vor- und Nachteile diese Integration in die jeweilige Architektur mit sich bringt. Für welche der Architekturen ist die Integration des E-Mail-Notifikationssystems am einfachsten?